

中南大学

《数据结构》课程实验 实验报告

实验题目 线性表的操作

专业班级 软件工程 2005 班

学 号 8209200504

姓 名 李均浩

实验成绩:

批阅教师:

2021 年 3 月 27 日

一、需求分析

1.程序任务

使用顺序存储结构以及链接存储结构，实现线性表的基本操作，插入、删除、查找，以及线性表合并等运算。之后利用线性表的各种基本操作实现一元 n 次多项式的排序，以及加法运算。

2.输入以及输出的形式

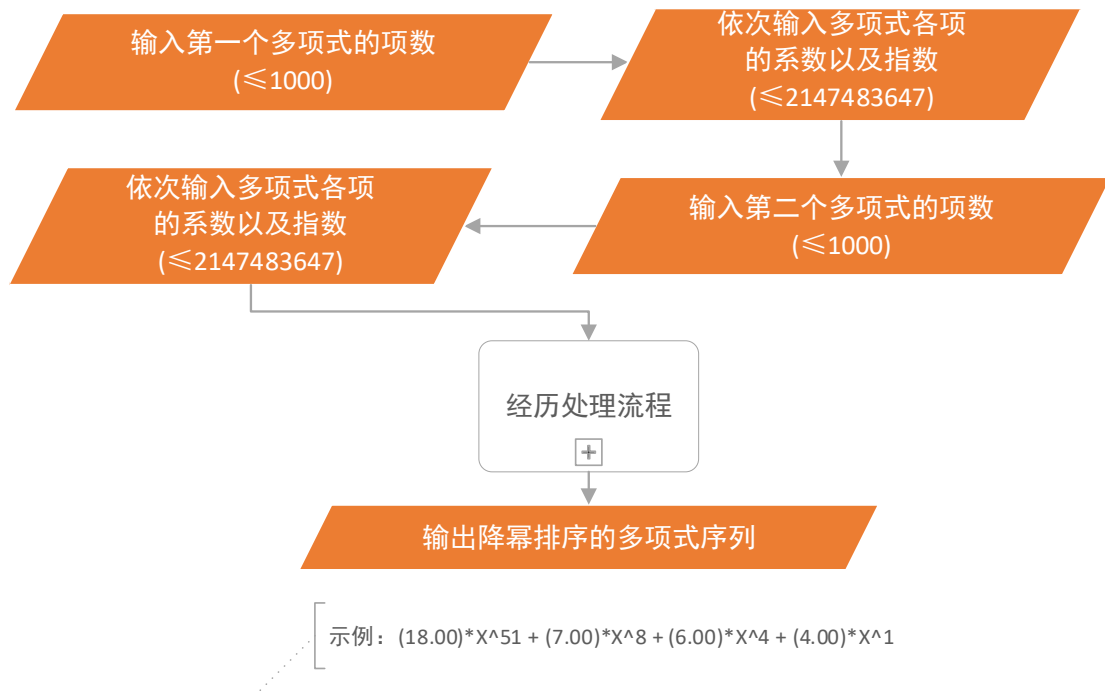


图1 程序输入输出形式

3.程序功能

实现多项式的相加，降幂排列，以及同指数项合并。

4.测试数据

(1)正确输入

(a) 5 1 4 21 54 7 8 9 1 4 2 2 1 88 3 22

预期输出： $(54.00)*X^{21} + X^9 + (8.00)*X^7 + (2.00)*X^4 + (22.00)*X^3 + (92.00)*X^1$

(b) 4 5 1 4 7 1 4 5 3 2 2 1 1 2

预期输出： $(4.00)*X^5 + (7.00)*X^4 + X^2 + (6.00)*X^1$

(2)非法输入

(a) [

预期输出： Invalid Input: No error

退出代码： 666

(b) 4 2 4 \

预期输出： Invalid Input: No error

退出代码： 666

二、概要设计

1.抽象数据类型定义:

ADT Polynomial{

数据对象: $\text{exp}=\{x \mid x \in \mathbb{Z}\}$ [3]

数据关系: $R=\{<\text{coe},\text{exp}>\}$

基本操作:

MakeNode(Link* p, ElemType e)

操作结果: 创建一个节点。

FreeNode(Link p, LinkList l)

初始条件: 节点 P 存在。

操作结果: 释放一个节点。

InitList(LinkList& l)

操作结果: 初始化一个链表。

DestroyList(LinkList& l)

初始条件: 链表 L 存在。

操作结果: 摧毁一个链表。

ClearList(LinkList& l)

初始条件: 链表 L 存在。

操作结果: 清空链表。

InsFirst(Link headNode, Link insertNode)

初始条件: 链表 L 存在。

操作结果: 在 L 的头节点前插入一个新的节点。

DelFirst(Link headNode, Link& q)

初始条件: 链表 L 存在。

操作结果: 删除 L 的头节点。

Append(LinkList& l, Link s)

初始条件: 链表 L 存在。

操作结果: 在链表的末尾插入一个节点。

Remove(LinkList& l, Link& q)

初始条件: 链表 L 存在, 节点 P 存在。

操作结果: 删除 P 节点。

InsBefore(LinkList& l, Link& p, Link s)

初始条件: 链表 L 存在, 节点 P 存在。

操作结果: 在 P 之前插入一个节点。

InsAfter(LinkList& l, Link& p, Link s)

初始条件: 链表 L 存在, 节点 P 存在。

操作结果: 在 P 之后插入一个节点。

SetCurElem(Link& p, ElemType e)

初始条件: 链表 L 存在, 节点 P 存在。

操作结果: 对节点 P 的数据元素赋值。

ElemType GetCurElem(Link p)

初始条件：链表 L 存在，节点 P 存在。

操作结果：获取节点 P 的数据元素的值。

ListEmpty(LinkList l)

初始条件：链表 L 存在。

操作结果：判断链表是否为空表并返回真假值。

ListLength(LinkList l)

初始条件：链表 L 存在。

操作结果：返回链表的元素个数。

GetHead(LinkList l)

初始条件：链表 L 存在。

操作结果：返回链表 L 的头指针。

GetLast(LinkList l)

初始条件：链表 L 存在。

操作结果：返回链表 L 的尾指针。

PriorPos(LinkList l, Link p)

初始条件：链表 L 存在，节点 P 存在。

操作结果：返回指向节点 P 前驱的指针。

NextPos(LinkList l, Link p)

初始条件：链表 L 存在，节点 P 存在。

操作结果：返回指向节点 P 后继的指针。

LocatePos(LinkList l, int index, Link& p)

初始条件：链表 L 存在。

操作结果：返回指向链表中第 index 个节点的指针。

LocateElem(LinkList l, ElemType e, Status(*compare)(ElemType, ElemType))

初始条件：链表 L 存在。

操作结果：在链表 L 中寻找储存满足 compare()条件的节点位置，并返回指向该节点的指针。

ListTraverse(LinkList l, Status (*visit)(LinkList l))

初始条件：链表 L 存在。

操作结果：遍历链表。

}ADT Poly

2.主程序的流程

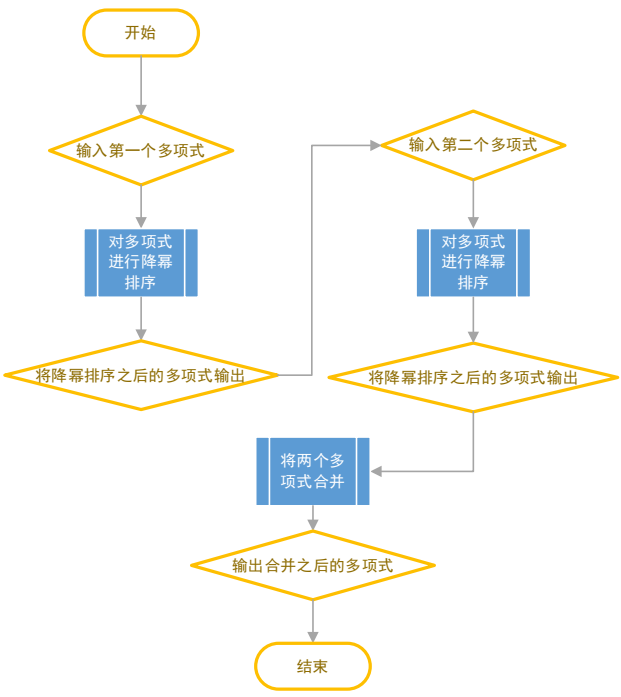


图2 主程序的流程

三、详细设计

1.模块伪码

```
(1)
InitList Begin
创建Link*类型变量 p;
申请大小为20的内存区域, 转换为Link*类型后并将地址赋给p;
if (p不是空指针) {
    p->next ← NULL;
    l.head ← p;
    l.tail ← p;
    l.elemAmount ← 0;
    return SUCCESS信号;
}
else
    输出错误信息("无法初始化链表");
End

(2)
GetPoly Begin
Link last = l.head;
Link cur;
for i ← 0 to 链表1的长度{
```

```

if (cur == NULL)
{
    输出错误信息("ERROR");
    return NULL;
}
else
{
    char check[100];
    输出("请输入第i+1项的指数: ");
    输入至 check;
    if (check不是数字串) {
        输出错误信息("Invalid Input");
        退出程序(报错代码:INVALID_INPUT);
    }
    else
        cur->exp ← atoi(check);
    printf("请输入第i + 1项的系数: ");
    输入至 check;
    if (check不是数字串) {
        输出错误信息("Invalid Input");
        退出程序(报错代码:INVALID_INPUT);
    }
    else
        cur->coe = atoi(check);
}
last ← cur;
cur ← (申请一块内存区域, 转换为Link类型, 大小为20);
last->next ← cur;
}
l.tail ← last;
释放(cur);
return SUCCESS 信号;
}

```

End

(3)

PrintPoly Begin

Link node ← l.head->next;

for i ← 0 to length - 1

```

{
    if (node->coe == 0)
        跳过本次循环;
    else {
        if (node->coe == 1)
            输出(" X^当前节点存储的指数 +");
    }
}

```

```

        else
            输出(" (系数)*X^指数 +");
    }
    node指向node的后继;
}

```

```

if (node->coe != 0)
{
    if (node->coe == 1)
        输出(" X^当前节点的指数");
    else
        输出(" (系数)*X^指数");
}
return SUCCESS信号;
End

```

(4)

SortPoly Begin

建立bool变量 isChanged, 标记是否进行了交换, 初始化标记为真;

```

bool isEqual ← true;
int equalCase ← 0;
建立指向链表L头指针的Link类型指针变量 ← l.head;

```

//降幂排序所有的项

```

while (进行了交换)
{
    标记:没有进行交换;
    node ← l.head->next;
    建立Link类型指针last ← l.head;

    for i ← 0 to length
    {
        if (node->exp < node->next->exp) {
            交换node节点与node后继节点的位置
            标记: 已经交换
            last向后移动;
        }
        else {
            last ← node;
            node ← node->next;
        }
    }
}
}

```

```

node ← l.head->next;

for i ← 0 to length - 1
{
    if (node->exp == node->next->exp) {
        node->coe ← node->next->coe + node->coe;
        LinkListNode* tmp ← node->next;
        node->next ← tmp->next;
        equalCase++;
        释放(tmp);
    }
    else
        node ← node->next;
}
length -= equalCase;
return SUCCESS信号;
End

```

(5)

```

MergePoly Begin
    初始化一个新的链表(poly_3);
    poly_3.head->next ← poly_1.head->next;
    LinkListNode* p ← poly_3.head->next;

    for i ← 0 to ElemAmount_1-1
    {
        p ← p->next;
    }
    p->next ← poly_2.head->next;
    ElemAmount_3 ← ElemAmount_1 + ElemAmount_2;
    将poly_3按降幂排列;
    return SUCCESS信号;
End

```


2.函数调用关系图

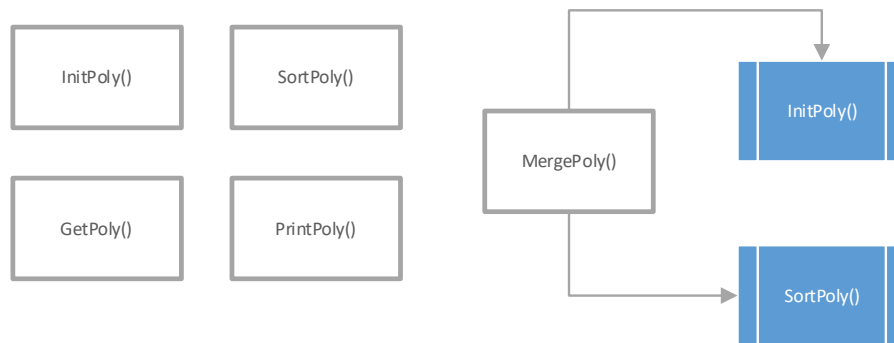


图3 函数调用关系图

四、调试分析

1.问题复现

(1)0xDDDDDDDD 读取访问权限冲突

(a)错误信息

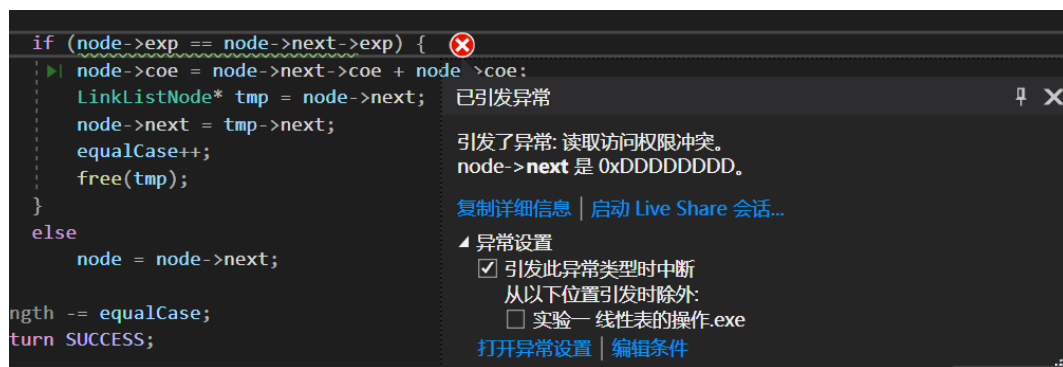


图4 0xDDDDDDDD 报错快照

(b)错误引发源

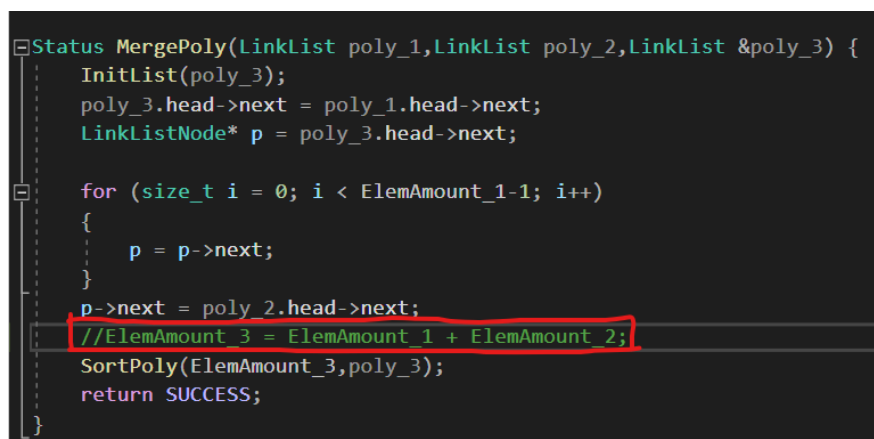


图5 存在问题的源代码

在 MergePoly(用于合并两个多项式)函数中，合并形成的新链表的元素个数没有更新，导致在 PrintPoly(用于输出多项式)函数中指针越界，导致了内存访问冲突。

2. 算法的时空分析

(1)改进设想

在 SortPoly 函数中实质上使用了冒泡排序的算法，时间复杂度为 $O(n^2)$ 在运行效率上较低，在大量数据下，消耗时间会较长。改进时，可将其改为其它时间复杂度更低的排序算法来提高程序的运行效率。

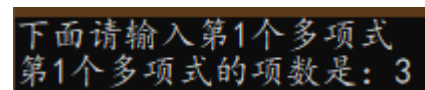
3. 经验与体会

在本次实验中，出现的主要问题来自于指针的错误，在编写程序的时候，有时会将指针指向了界外的位置，导致了内存的读取冲突。在出现 bug，改正 bug 的往复中，对链表及其应用有了更深入的理解。在链表的遍历程序段中，for 循环的终止点要仔细确认；函数中新建的指针变量要初始化到正确的位置；涉及到元素个数的情况要及时按照判断条件更新元素个数。链表中数据的读取虽然不同数组一样可以随机读写，但它能将大量的数据存储在内存中，而不必占用连续的内存空间，让内存空间的使用效率更高了，在以后的练习和实践中也经常尝试去使用。

五、用户使用说明

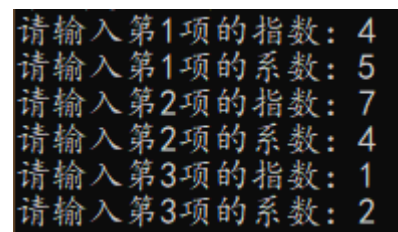
1.输入第一个多项式

2.输入多项式中包含的项数(整数, $0 \leq x \leq 214783647$)



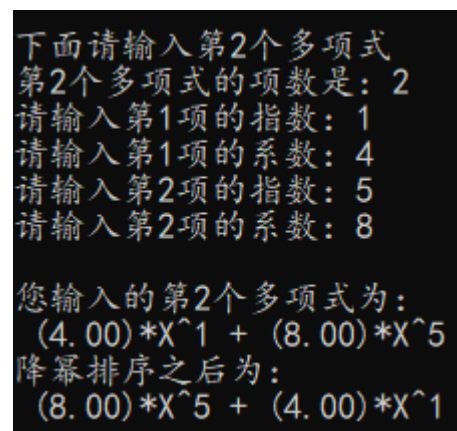
```
下面请输入第1个多项式
第1个多项式的项数是: 3
```

3.依次输入各项的指数(整数, $0 \leq x \leq 214783647$)以及系数(浮点数, $3.40282e+038$)



```
请输入第1项的指数: 4
请输入第1项的系数: 5
请输入第2项的指数: 7
请输入第2项的系数: 4
请输入第3项的指数: 1
请输入第3项的系数: 2
```

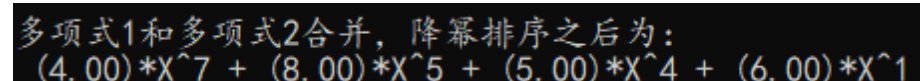
4.重复上述步骤以输入第二个多项式



```
下面请输入第2个多项式
第2个多项式的项数是: 2
请输入第1项的指数: 1
请输入第1项的系数: 4
请输入第2项的指数: 5
请输入第2项的系数: 8

您输入的第2个多项式为:
(4.00)*X^1 + (8.00)*X^5
降幂排序之后为:
(8.00)*X^5 + (4.00)*X^1
```

5.得到第一个多项式与第二个多项式的降幂排序之后的合并结果



```
多项式1和多项式2合并, 降幂排序之后为:
(4.00)*X^7 + (8.00)*X^5 + (5.00)*X^4 + (6.00)*X^1
```

六、测试结果

(1)输入: 5 1 12 45 21 2 5 12 75 1 2 4 12 4 5 4 1 8 6 2

输出:

```
多项式1和多项式2合并, 降幂排序之后为:  
(21.00)*X^45 + (79.00)*X^12 + (2.00)*X^6 + (4.00)*X^5 + (5.00)*X^2 + (22.00)*X^1
```

(2)输入: 5 1 4 5 21 1 3 4 5 1 5 4 1 2 5 4 5 1 1 5 5 1 2 3

输出:

```
多项式1和多项式2合并, 降幂排序之后为:  
X^21 + (12.00)*X^5 + (5.00)*X^4 + (4.00)*X^3 + (3.00)*X^2 + (7.00)*X^1
```

(3)输入: 5 1 2 1 4 7 8 9 1 2 2 4 5 4 4 5

输出:

```
多项式1和多项式2合并, 降幂排序之后为:  
(9.00)*X^8 + (5.00)*X^5 + (12.00)*X^4 + (5.00)*X^2 + (2.00)*X^1
```

(4)输入: 8 4 5 6 5 1 2 5 7 8 2 1 4 5

输出:

```
多项式1和多项式2合并, 降幂排序之后为:  
(4.00)*X^8 + (8.00)*X^7 + (7.00)*X^5 + (5.00)*X^4 + (6.00)*X^2
```

(5)输入: 5 4 8 8 8 8 5 1 2 4 7 5 2 1 4 7

输出:

```
多项式1和多项式2合并, 降幂排序之后为:  
(16.00)*X^8 + (5.00)*X^7 + (5.00)*X^5 + (7.00)*X^4 + (5.00)*X^2
```

(6)输入: \

输出:

```
第1个多项式的项数是: \  
Invalid Input: No error  
D:\数据结构\实验\线性表\线性表\Debug\实验一 线性表的操作.exe (进程 4456)已退出, 代码为 666。
```

(7)输入: 4 4 78 6 3 4 5 8 j4

输出:

```
Invalid Input: No error  
D:\数据结构\实验\线性表\线性表\Debug\实验一 线性表的操作.exe (进程 32300)已退出, 代码为 666。
```

(8)输入: 1]

输出:

```
下面请输入第1个多项式
第1个多项式的项数是: 1
请输入第1项的指数: ]
Invalid Input: No error

D:\数据结构\实验\线性表\线性表\Debug\实验一 线性表的操作.exe (进程 4828) 已退出, 代码为 666。
```

(9)输入: 4 5 ##

输出:

```
下面请输入第1个多项式
第1个多项式的项数是: 4
请输入第1项的指数: 5
请输入第1项的系数: ##
Invalid Input: No error

D:\数据结构\实验\线性表\线性表\Debug\实验一 线性表的操作.exe (进程 15356) 已退出, 代码为 666。
```

(10)输入: 3 4o 55 47mm 3n 9 1 5 5nj 1 4 5i 9 1 4rr 4ds 4q 2x

输出:

```
多项式1和多项式2合并, 降幂排序之后为:
(3.00)*X^47 + (2.00)*X^9 + X^5 + (66.00)*X^4
```

七、附录

```
#include <iostream>
#include <cstdio>
#include <malloc.h>
#include <process.h>

using namespace std;

#define SUCCESS 1
#define ERROR 0
#define INIT_SIZE 10
#define MAX_SIZE 100
#define TRUE 1
#define FALSE 0

//int可改为其它数据类型
typedef int ElemType;
typedef int Status;

typedef struct
{
    int length;
    size_t listSize;
    int* elemAddress;
}SeqList;

//初始化顺序线性表
Status InitList(SeqList& l)
{
    l.elemAddress = (ElemType*)malloc(INIT_SIZE); //为线性表L分配内存空间
    if (!l.elemAddress)
    {
        exit(OVERFLOW); //如果无法申请到内存空间，报错代码“OVERFLOW 3”
    }
    l.length = 0;
    l.listSize = INIT_SIZE;
    return SUCCESS;
}

//摧毁一个顺序线性表
Status DestroyList(SeqList& l)
{
    free(l.elemAddress);
}
```

```

        l.elemAddress = NULL;
        l.length = 0;
        l.listSize = 0;
        return SUCCESS;
    }

//清空一个顺序线性表
Status ClearList(SeqList& l)
{
    l.length = 0;
    return SUCCESS;
}

//判断顺序线性表是否为空
Status ListEmpty(SeqList& l)
{
    if (l.length == 0)
        return TRUE;
    else
        return FALSE;
}

//获取顺序线性表中第i个元素数据的值
Status GetElem(SeqList l, int i, ElemType& e)
{
    e = l.elemAddress[i];
    return SUCCESS;
}

//在顺序线性表中寻找储存满足compare()条件的位置
int LocateElem(SeqList l, ElemType e, Status compare(ElemType, ElemType))
{
    int i = 0;
    for (i = 0; i <= l.length; ++i)
    {
        if (compare(l.elemAddress[i], e))
        {
            break;
        }
    }
    if (i < l.length)
        return i;
    else
        return ERROR;
}

```

```

}

//获取顺序线性表传入元素的前一个元素
Status PriorElem(SeqList l, ElemType curElem, ElemType& preElem)
{
    if (l.elemAddress[0] != preElem)
    {
        int i = 1;
        while (i < l.length && l.elemAddress[i] != curElem)
        {
            i++;
        }
        preElem = l.elemAddress[i - 1];
        return SUCCESS;
    }
    return ERROR;
}

```

```

//获取顺序线性表传入元素的下一个元素
Status NextElem(SeqList l, ElemType curElem, ElemType& nextElem)
{
    if (l.elemAddress[l.length - 1] != curElem)
    {
        int i = 0;
        while (i < l.length && l.elemAddress[i] != curElem)
        {
            i++;
        }
        nextElem = l.elemAddress[i + 1];
        return SUCCESS;
    }
    else
        return ERROR;
}

```

```

//在顺序线性表的第index个位置插入新元素
Status ListInsert(SeqList& l, int index, ElemType& e)
{
    for (int i = l.length - 1; i >= index; --i)
    {
        l.elemAddress[i + 1] = l.elemAddress[i];
    }
    l.elemAddress[index] = e;
    l.length++;
}

```

```

    return SUCCESS;
}

//删除顺序线性表中的第index个元素
Status ListDelete(SeqList& l, int index, ElemType& e)
{
    e = l.elemAddress[index];
    for (int i = index; i < l.length; ++i)
    {
        l.elemAddress[i] = l.elemAddress[i + 1];
    }
    l.length--;
    return SUCCESS;
}

//归并La,Lb至Lc
Status ListMerge(SeqList la, SeqList lb, SeqList& lc)
{
    ElemType* a_tail = la.elemAddress + la.length - 1;
    ElemType* b_tail = lb.elemAddress + lb.length - 1;

    ElemType* pa = la.elemAddress;
    ElemType* pb = lb.elemAddress;

    ElemType* pc = (ElemType*)malloc(la.length + lb.length);

    while (pa <= a_tail && pb <= b_tail)
    {
        if (*pa <= *pb)
        {
            *pc = *pa;
            pc++;
            pa++;
        }
        else
        {
            *pc = *pb;
            pc++;
            pb++;
        }
    }
    while (pa <= a_tail)
    {
        *pc = *pa;
    }
}

```



```

        pc++;
        pa++;
    }
    while (pb <= b_tail)
    {
        *pc = *pb;
        pc++;
        pb++;
    }
}

```

源代码1 基本操作-数组实例.cpp

```

#include <iostream>
#include <cstdio>
#include <malloc.h>
#include <process.h>

using namespace std;

#define SUCCESS 1
#define ERROR 0
#define INIT_SIZE 10
#define MAX_SIZE 100
#define TRUE 1
#define FALSE 0

//int可改为其它数据类型
typedef int ElemType;
typedef int Status;

//节点类型
typedef struct LinkListNode
{
    ElemType data;
    struct LinkListNode* next;
}*Link, *Position;

//链表类型
typedef struct SinglyLinkList
{
    Link head, tail;
    int elemAmount;
}LinkList;

```

```

Status MakeNode(Link* p, ElemType e)
{
    *p = (Link)malloc(sizeof(LinkListNode));
    if (p == NULL)
    {
        perror("ERROR");
        return NULL;
    }
    else
    {
        (*p)->data = e;
        return SUCCESS;
    }
}

```

```

void FreeNode(Link p, LinkList l)
{
    Link search = l.head;
    while (search != NULL && search->next != p)
    {
        search = search->next;
    }
    if (search->next == p)
    {
        search->next = p->next;
    }
    free(p);
}

```

//初始化链表

```

Status InitList(LinkList& l)
{
    Link p;
    p = (Link)malloc(sizeof(Link));
    if (p != NULL)
    {
        p->next = NULL;
        l.head = p;
        l.tail = p;
        l.elemAmount = 0;
        return SUCCESS;
    }
    else
        perror("Cannot Initial LinkList");
}

```

```

t");
}

//摧毁一个链表
Status DestroyList(LinkList& l)
{
    Link p = l.head;
    Link tmp;
    while (p != NULL)
    {
        tmp = p;
        p = p->next;
        free(tmp);
    }
    l.head = NULL;
    l.tail = NULL;
    l.elemAmount = NULL;
    return SUCCESS;
}

//链表清空
Status ClearList(LinkList& l)
{
    l.elemAmount = 0;
    return SUCCESS;
}

//将一个节点插入到头节点前
Status InsFirst(Link headNode, Link insertNode)
{
    insertNode->next = headNode;
    return SUCCESS;
}

//删除头节点
Status DelFirst(Link headNode, Link& q)
{
    q = headNode->next;
    free(headNode);
    return SUCCESS;
}

//在链表的尾部插入一个节点
Status Append(LinkList& l, Link s)

```

```

{
    l.tail->next = s;
    return SUCCESS;
}

//删除节点
Status Remove(LinkList& l, Link& q)
{
    q = l.tail;
    Link p = l.head;
    while (p->next != l.tail)
    {
        p = p->next;
    }
    l.tail = p;
    return SUCCESS;
}

//在p节点之前插入一个节点
Status InsBefore(LinkList& l, Link& p, Link s)
{
    Link tmp = l.head;
    while (tmp->next != p)
    {
        tmp = tmp->next;
    }
    tmp->next = s;
    s->next = p;
    p = s;
    return SUCCESS;
}

//在p节点之后插入一个节点
Status InsAfter(LinkList& l, Link& p, Link s)
{
    Link tmp = p->next;
    p->next = s;
    s->next = tmp;
    p = s;
    return SUCCESS;
}

//为节点p中的数据域赋值
Status SetCurElem(Link& p, ElemType e)

```

```

{
    p->data = e;
    return SUCCESS;
}

//获取p节点中存储的数据值
ElemType GetCurElem(Link p)
{
    return p->data;
}

//判断链表是否为空
Status ListEmpty(LinkList l)
{
    if (!l.head)
    {
        perror("The LinkList is not exist!");
        return ERROR;
    }
    if (l.head->next == NULL)
        return TRUE;
    else
        return FALSE;
}

//求取链表的元素个数
int ListLength(LinkList l)
{
    if (!l.head)
    {
        perror("The LinkList is not exist!");
        return ERROR;
    }
    return l.elemAmount;
}

//获取头指针
Position GetHead(LinkList l)
{
    return l.head;
}

//获取尾指针
Position GetLast(LinkList l)

```

```

{
    return l.tail;
}

//获取p节点的前驱
Position PriorPos(LinkList l, Link p)
{
    if (p == l.head)
        return ERROR;
    else
    {
        Link tmp = l.head;
        while (tmp->next != p)
        {
            tmp = tmp->next;
        }
        return tmp;
    }
}

//获取p节点的后继
Position NextPos(LinkList l, Link p)
{
    if (p == l.tail)
        return ERROR;
    else
    {
        return p->next;
    }
}

//返回指向链表中第index个节点的指针
Status LocatePos(LinkList l, int index, Link& p)
{
    if (!(index >= '0' && index <= '9') || index > l.elemAmount)
    {
        perror("Invalid index figure");
        return ERROR;
    }
    Link tmp = l.head;
    for (int i = 1; i < index; ++i)
    {
        tmp = tmp->next;
    }
}

```

```

    p = tmp;
    return SUCCESS;
}

//在链表L中寻找储存满足compare()条件的节点位置，并返回指向该节点的指针
Position LocateElem(LinkList l, ElemType e, Status(*compare)(ElemType, ElemType))
{
    Link p = l.head;
    while (p != NULL && !(compare((*p).data), e))
    {
        p = p->next;
    }
    return p;
}

//遍历链表
Status ListTraverse(LinkList l, Status (*visit)(LinkList l))
{
    Link p=l.head;
    while (p!=NULL)
    {
        if (visit(l))
            p = p->next;
        else
            return ERROR;
    }
    return SUCCESS;
}

```

源代码2 基本操作-链表实例.cpp

```

#include <iostream>
#include <stdio>
#include <malloc.h>

#define SUCCESS 1
#define ERROR 0
#define INIT_SIZE 100
#define MAX_SIZE 200

using namespace std;

typedef int Status;

```

```

struct Poly {
    int exp;
    float coe;
}poly_1[INIT_SIZE], poly_2[INIT_SIZE], poly_3[MAX_SIZE];

int ElemAmount_3 = 0;

//获取多项式的系数和指数
Status GetPoly(int length, Poly* poly) {
    for (size_t i = 0; i < length; i++)
    {
        printf("请输入第%d项的指数: ", i + 1);
        scanf_s("%d", &poly[i].exp);

        printf("请输入第%d项的系数: ", i + 1);
        scanf_s("%f", &poly[i].coe);
    }
    return SUCCESS;
}

//输出多项式
Status PrintPoly(int length, Poly* poly) {
    for (size_t i = 0; i < length - 1; i++)
    {
        if (poly[i].coe == 0)
            continue;
        else {
            if (poly[i].coe == 1)
                printf(" X^%d +", poly[i].exp);
            else
                printf(" (%.2f)*X^%d +", poly[i].coe, poly[i].exp);
        }
    }
    if (poly[length - 1].coe != 0)
    {
        if (poly[length - 1].coe == 1)
            printf(" X^%d", poly[length - 1].exp);
        else
            printf(" (%.2f)*X^%d", poly[length - 1].coe, poly[length - 1].exp);
    }
    return SUCCESS;
}

//降幂排序

```



```

Status SortPoly(int& length, Poly* poly) {
    int equalCase = 0;
    for (size_t i = 0; i < length - 1; i++)
    {
        for (size_t j = i + 1; j < length; j++)
        {
            if (poly[i].exp == poly[j].exp && poly[i].exp != -1)
            {
                poly[i].coe += poly[j].coe;
                poly[j].exp = -1;
                poly[j].coe = 0;
                equalCase++;
            }
            if (poly[i].exp < poly[j].exp) {
                int exp_temp;
                exp_temp = poly[j].exp;
                poly[j].exp = poly[i].exp;
                poly[i].exp = exp_temp;
                float coe_temp;
                coe_temp = poly[j].coe;
                poly[j].coe = poly[i].coe;
                poly[i].coe = coe_temp;
            }
        }
    }
    length -= equalCase;
    return SUCCESS;
}

```

//将两个多项式合并，按照降幂顺序排列

```

Status MergePoly(int ElemAmount_1, int ElemAmount_2, Poly* poly_1, Poly* poly_2,
Poly* poly_3) {
    int i = 0, j = 0, k = 0;
    while (i < ElemAmount_1 && j < ElemAmount_2)
    {
        if (poly_1[i].exp == poly_2[j].exp) {
            poly_3[k].coe = poly_1[i].coe + poly_2[j].coe;
            poly_3[k].exp = poly_1[i].exp;
            i++;
            j++;
            k++;
        }
        if (poly_1[i].exp > poly_2[j].exp) {
            poly_3[k].exp = poly_1[i].exp;

```

```

        poly_3[k].coe = poly_1[i].coe;
        i++;
        k++;
    }
    if (poly_1[i].exp < poly_2[j].exp)
    {
        poly_3[k].exp = poly_2[j].exp;
        poly_3[k].coe = poly_2[j].coe;
        j++;
        k++;
    }
}
if (i == ElemAmount_1) {
    for (; j < ElemAmount_2; j++)
    {
        poly_3[k + 1].coe = poly_2[j].coe;
        poly_3[k + 1].exp = poly_2[j].exp;
        k++;
    }
}
else {
    for (; i < ElemAmount_1; i++)
    {
        poly_3[k + 1].coe = poly_1[i].coe;
        poly_3[k + 1].exp = poly_1[i].exp;
        k++;
    }
}
ElemAmount_3 = k;
return SUCCESS;
}

int main() {
    int ElemAmount_1 = 0;
    cout << "下面请输入第1个多项式" << endl;
    cout << "第一个多项式的项数是: ";
    cin >> ElemAmount_1;
    GetPoly(ElemAmount_1, poly_1);
    cout << "\n您输入的第1个多项式为: \n";
    PrintPoly(ElemAmount_1, poly_1);
    SortPoly(ElemAmount_1, poly_1);
    cout << endl << "降幂排序之后为: " << endl;
    PrintPoly(ElemAmount_1, poly_1);
}

```

```

        cout << endl << endl;

        int ElemAmount_2 = 0;
        cout << "下面请输入第2个多项式" << endl;
        cout << "第2个多项式的项数是: ";
        cin >> ElemAmount_2;
        GetPoly(ElemAmount_2, poly_2);
        cout << "\n您输入的第2个多项式为: \n";
        PrintPoly(ElemAmount_2, poly_2);
        SortPoly(ElemAmount_2, poly_2);
        cout << endl << "降幂排序之后为: " << endl;
        PrintPoly(ElemAmount_2, poly_2);

        MergePoly(ElemAmount_1, ElemAmount_2, poly_1, poly_2, poly_3);

        printf("\n\n多项式1和多项式2合并之后为: \n");
        PrintPoly(ElemAmount_3, poly_3);

        return 0;
}

```

源代码3 一元 n 次多项式-数组实例.cpp

```

#include <iostream>
#include <string>
#include <cstdio>
#include <malloc.h>

using namespace std;

#define SUCCESS 1
#define ERROR 0
#define INIT_SIZE 100
#define MAX_SIZE 200
#define TRUE 1
#define FALSE 0
#define INVALID_INPUT 666

//Status作为函数返回值的类型，返回函数工作的状态
typedef int Status;

//节点类型
typedef struct LinkListNode
{

```

```

//data
int exp;
float coe;

//index
struct LinkListNode* next;
}*Link, * Position;

//链表类型
typedef struct SinglyLinkList
{
    Link head, tail;
    int elemAmount;
}LinkList;

int ElemAmount_1 = 0;
int ElemAmount_2 = 0;
int ElemAmount_3 = 0;

//初始化一个链表，其中包含了头指针，尾指针和元素个数变量
Status InitList(LinkList& l)
{
    Link p;
    p = (Link)malloc(20);
    if (p != NULL)
    {
        p->next = NULL;
        l.head = p;
        l.tail = p;
        l.elemAmount = 0;
        return SUCCESS;
    }
    else
        perror("Cannot Initial LinkList");
}

//判断一串字符串是否为数字串并返回布尔值
bool IsNumber(const char* str)
{
    double aa;
    int nn = sscanf_s(str, "%lf", &aa);
    return nn != 0;
}

```

```

//获取多项式的系数和指数
Status GetPoly(int length, LinkList l) {
    Link last = l.head;
    Link cur;
    cur = (Link)malloc(20);
    last->next = cur;

    for (size_t i = 0; i < length; i++)
    {
        if (cur == NULL)
        {
            perror("ERROR");
            return NULL;
        }
        else
        {
            char check[100];
            printf("请输入第%d项的指数: ", i + 1);
            cin >> check;
            if (!IsNumber(check)) {
                perror("Invalid Input");
                exit(INVALID_INPUT);
            }
            else
                cur->exp = atoi(check);
            printf("请输入第%d项的系数: ", i + 1);
            cin >> check;
            if (!IsNumber(check)) {
                perror("Invalid Input");
                exit(INVALID_INPUT);
            }
            else
                cur->coe = atoi(check);
        }
        last = cur;
        cur = (Link)malloc(20);
        last->next = cur;
    }
    l.tail = last;
    free(cur);
    return SUCCESS;
}

```

```

//输出多项式

```

```

Status PrintPoly(int length, LinkList l) {
    Link node = l.head->next;
    for (size_t i = 0; i < length - 1; i++)
    {
        if (node->coe == 0)
            continue;
        else {
            if (node->coe == 1)
                printf(" X^%d +", node->exp);
            else
                printf(" (%.2f)*X^%d +", node->coe, node->exp);
        }
        node = node->next;
    }

    if (node->coe != 0)
    {
        if (node->coe == 1)
            printf(" X^%d", node->exp);
        else
            printf(" (%.2f)*X^%d", node->coe, node->exp);
    }
    return SUCCESS;
}

```

//将多项式按照降幂排序重新排列

```

Status SortPoly(int& length, LinkList& l) {
    bool isChanged = true;
    bool isEqual = true;
    int equalCase = 0;
    Link node = l.head;

    //降幂排序所有的项
    while (isChanged)
    {
        isChanged = false;

        node = l.head->next;
        Link last = l.head;

        for (size_t i = 0; i < length; i++)
        {
            if (node->exp < node->next->exp) {
                Link temp;

```

```

        temp = node->next;
        node->next = temp->next;
        last->next = temp;
        temp->next = node;
        isChanged = true;
        last = last->next;
    }
    else {
        last = node;
        node = node->next;
    }
}

//以下代码段用于合并，并释放一个指数值相同的节点
node = l.head->next;
for (size_t i = 0; i < length - 1; i++)
{
    if (node->exp == node->next->exp) {
        node->coe = node->next->coe + node->coe;
        LinkListNode* tmp = node->next;
        node->next = tmp->next;
        equalCase++;
        free(tmp);
    }
    else
        node = node->next;
}
length -= equalCase;
return SUCCESS;
}

//将两个多项式合并，按照降幂顺序排列
Status MergePoly(LinkList poly_1, LinkList poly_2, LinkList& poly_3) {
    InitList(poly_3);
    poly_3.head->next = poly_1.head->next;
    LinkListNode* p = poly_3.head->next;

    for (size_t i = 0; i < ElemAmount_1 - 1; i++)
    {
        p = p->next;
    }
}

```

```

    p->next = poly_2.head->next;
    ElemAmount_3 = ElemAmount_1 + ElemAmount_2;
    SortPoly(ElemAmount_3, poly_3);
    return SUCCESS;
}

int main() {
start:
    char* check = new char[50]; //用于缓存输入的项数
    cout << "下面请输入第1个多项式" << endl;
    cout << "第1个多项式的项数是: ";
    cin >> check;
    if (!IsNumber(check)) { //检测是否为非法输入
        perror("Invalid Input");
        exit(INVALID_INPUT);
    }
    else
        ElemAmount_1 = atoi(check);
    LinkList poly_1;
    InitList(poly_1);
    GetPoly(ElemAmount_1, poly_1);
    cout << "\n您输入的第1个多项式为: \n";
    PrintPoly(ElemAmount_1, poly_1);
    SortPoly(ElemAmount_1, poly_1);
    cout << endl << "降幂排序之后为: " << endl;
    PrintPoly(ElemAmount_1, poly_1);

    cout << "\n\n下面请输入第2个多项式" << endl;
    cout << "第2个多项式的项数是: ";
    cin >> check;
    if (!IsNumber(check)) {
        perror("Invalid Input");
        exit(INVALID_INPUT);
    }
    else
        ElemAmount_2 = atoi(check);
    delete check;
    LinkList poly_2;
    InitList(poly_2);
    GetPoly(ElemAmount_2, poly_2);
    cout << "\n您输入的第2个多项式为: \n";
    PrintPoly(ElemAmount_2, poly_2);
    SortPoly(ElemAmount_2, poly_2);
    cout << endl << "降幂排序之后为: " << endl;

```



```
PrintPoly(ElemAmount_2, poly_2);

LinkedList poly_3;
MergePoly(poly_1, poly_2, poly_3);

printf("\n\n多项式1和多项式2合并，降幂排序之后为： \n");
PrintPoly(ElemAmount_3, poly_3);
goto start;
return 0;

}
```

源代码4 一元 n 次多项式-链表实例.cpp